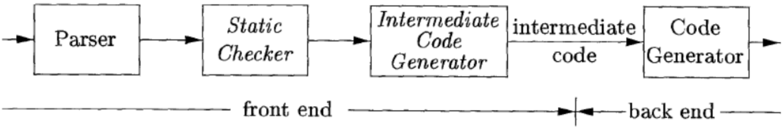
Intermediate-Code Generation

Intermediate-Code Generation



Three-Address Code

- In three-address code, there is at **most one operator** on the **right side of an instruction**
- Thus a **source-language expression like x+y*z** might be translated into the sequence of three-address instructions

$$t_1 = y * z$$
$$t_2 = x + t_1$$

Common three-address instructions

1. Assignment instructions of the form $x = y \ op \ z$, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \ y$, where *op* is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number.

3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.

4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (`<`, `==`, `>=`, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y. If not, the three-address instruction following `if` x *relop* y `goto` L is executed next, in sequence.

7. Procedure calls and returns are implemented using the following instructions: `param` x for parameters; `call` p, n and y = `call` p, n for procedure and function calls, respectively; and `return` y, where y, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

`param` x_1
`param` x_2
...
`param` x_n
`call` p, n

generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$. The integer n, indicating the number of actual parameters in "`call` p, n,"

8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$. The instruction $x = y[i]$ sets x to the value in the location i memory units beyond location y. The instruction $x[i] = y$ sets the contents of the location i units beyond x to the value of y.

Common three-address instructions

```
do i = i+1; while (a[i] < v);
```

L: $t_1 = i + 1$
$i = t_1$
$t_2 = i * 8$
$t_3 = a [t_2]$
if $t_3 < v$ goto L

Data structures for TAC

Quadruples

A *quadruple* (or just "*quad*") has four fields, which we call *op*, arg_1, arg_2, and *result*. The *op* field contains an internal code for the operator. For instance, the three-address instruction $x = y + z$ is represented by placing + in *op*, y in arg_1, z in arg_2, and x in *result*. The following are some exceptions to this rule:

1. Instructions with unary operators like x = `minus` y or $x = y$ do not use arg_2. Note that for a copy statement like $x = y$, *op* is =, while for most other operations, the assignment operator is implied.

2. Operators like `param` use neither arg_2 nor *result*.

3. Conditional and unconditional jumps put the target label in *result*.

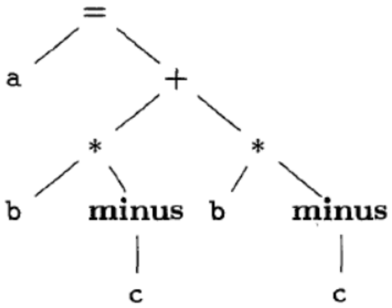
Data structures for TAC

Quadruples

Three-address code for the assignment `a = b*-c+b*-c;`

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code



(a) Syntax tree

Data structures for TAC
Quadruples

Three-address code for the assignment `a = b*-c+b*-c;`

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

Data structures for TAC
Triples

- A triple has **only three fields**, which we call **op, arg1, and arg2**

- Note that the **result field in Quad** is used primarily for temporary names.

- Using **triples**, we refer to the **result of an operation** x op y **by its position**, rather than by an explicit temporary name.

- Thus, instead of the **temporary t**, a triple representation would refer to **position (0)**.

- **Parenthesized numbers** represent pointers into the **triple structure** itself.

Data structures for TAC
Triples

Three-address code for the assignment `a = b * - c + b * - c ;`

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

Benefit of Quad over Triples

- A benefit of **quadruples** over triples can be seen in an **optimizing compiler**, where **instructions are often moved around**.
    - With quadruples, **if we move an instruction** that **computes a temporary t**, then the instructions that **use t** require **no change**.

- With **triples**, the **result** of an operation is referred to by **its position**

- So **moving an instruction** may require us to **change all references** to that result.

Indirect triples

- Consist of a **listing of pointers** to triples,
  - Rather than a listing of triples themselves.
- For example, use an **array *instruction*** to list **pointers to triples** in the desired order.

With indirect triples, an **optimizing compiler** can move an instruction by **reordering the instruction list**, without affecting the triples themselves.

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

	op	arg_1	arg_2
0	minus	c	
1	minus	c	
2	*	b	(0)
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Static Single-Assignment Form

Two distinctive aspects distinguish SSA from three-address code.
(a) The first is that **all assignments in SSA** are to **variables with distinct names**; hence the term static single-assignment.

$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$
(a) Three-address code.	(b) Static single-assignment form.

Static Single-Assignment Form

Two distinctive aspects distinguish SSA from three-address code.
(a) The first is that **all assignments in SSA** are to **variables with distinct names**; hence the term static single-assignment.
(b)

The same variable may be defined in two different control-flow paths in a program. For example, the source program

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

has two control-flow paths in which the variable x gets defined. If we use different names for x in the true part and the false part of the conditional statement, then which name should we use in the assignment y = x * a? Here is where the second distinctive aspect of SSA comes into play. SSA uses a notational convention called the ϕ-function to combine the two definitions of x:

$$\text{if (flag) } x_1 = -1; \text{ else } x_2 = 1;$$
$$x_3 = \phi(x_1, x_2);$$

Static Single-Assignment Form

Here, $\phi(\mathbf{x}_1, \mathbf{x}_2)$ has the value $\mathbf{x}_1$ if the control flow passes through the true part of the conditional and the value $\mathbf{x}_2$ if the control flow passes through the false part. That is to say, the ϕ-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the ϕ-function.

Major translation classes of Three address code generation

(a) Declaration statements (+ handling data type and storage)

(b) Expressions and statements

(a) Control flow statements

Declaration statement

Representing data types: Type Expressions

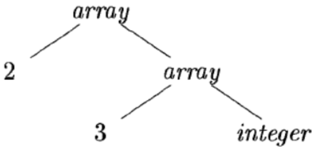
Types have structure, which we shall represent using type expressions.

- A **type expression is either a basic type** (*boolean, char, integer, float,* and *void*)

  or
- is formed by **applying an operator** called a **type constructor** to a type expression.
- A **type expression** can be formed by applying the **array type constructor** to a **number and a type expression**.

Declaration statement

- The array type int [2] [3] can be read as "array of 2 arrays of 3 integers each"
- Can be represented as a **type expression** array(2, array(3, integer)).
- This type is represented by the tree.



- The **operator array** takes **two parameters**, a number and a type.
  - Here the **type expression** can be formed by applying the **array type constructor** to a number and a type expression.

Declaration statement Example SDD

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Type Expressions

- Nonterminal T generates either a **basic type** or an **array type**.
- Nonterminal B generates one of the basic types int and float.
- T generates a basic type when C derives €.
- Otherwise, C generates array components consisting of a sequence of integers, each integer surrounded by brackets.

Declaration statement
Example SDD

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = integer$
$B \rightarrow \textbf{float}$	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

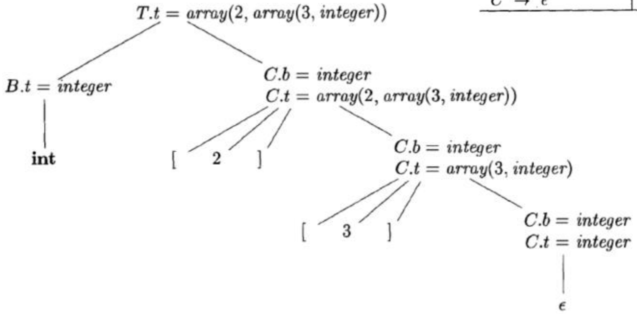
Type Expressions

- The nonterminals *B* and *T* have a synthesized attribute *t* representing a type.
- The nonterminal *C* has two attributes: an inherited attribute *b* and a synthesized attribute *t*.

Declaration statement Example SDD

input string int [2][3]

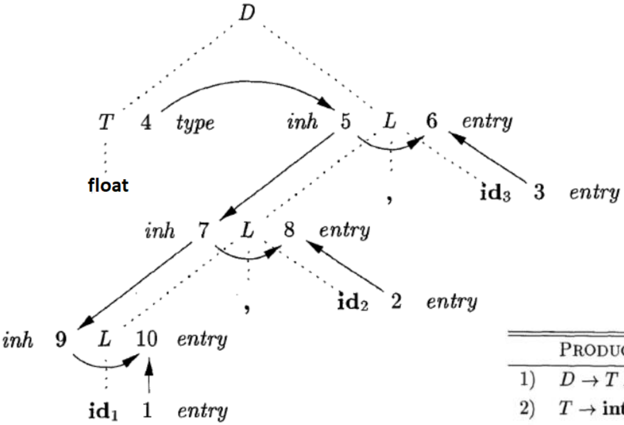
PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$



- The nonterminal *C* has two attributes: an inherited attribute *b* and a synthesized attribute *t.*
- The inherited *b* attributes pass a basic type down the tree, and the synthesized *t* attributes accumulate the result.

Declaration statement: Example SDD

$$\textbf{float}\ \textbf{id}_1\ ,\ \textbf{id}_2\ ,\ \textbf{id}_3$$



	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow$ **int**	$T.type =$ integer
3)	$T \rightarrow$ **float**	$T.type =$ float
4)	$L \rightarrow L_1$, **id**	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow$ **id**	$addType(\mathbf{id}.entry, L.inh)$

Symbol table

ST(global) *This is the Symbol Table for global symbols*

Name	Type	Initial Value	Size	Offset	Nested Table
d	float	2.3	8	0	null
i	int	null	4	8	null
w	*array*(10, int)	null	40	12	null
a	int	4	4	52	null
p	*ptr*(int)	null	4	56	null
b	int	null	4	60	null
func	*function*	null	0	64	ptr-to-ST(func)
c	char	null	1	64	null

Find the storage for each variable

More on Declaration statement

Data type + Storage layout

$$
\begin{array}{lcl}
D & \rightarrow & T \ \textbf{id} \ ; \ D \ | \ \epsilon \\
T & \rightarrow & B \ C \\
B & \rightarrow & \textbf{int} \ | \ \textbf{float} \\
C & \rightarrow & \epsilon \ | \ [\ \textbf{num} \ ] \ C
\end{array}
$$

- Simplified grammar that declares just **one name at a time**;
- We already explored the declarations with lists of names

Storage layout:

- **Relative address** of all the variables
- From the **type of a variable**, we can determine the **amount of storage** that will be needed for the variable at run time.
- At **compile time**, we can use these amounts to **assign each variable a relative address**.
- The **type** and **relative address** are saved in the **symbol-table entry** for the variable name.

More on Declaration statement

Data type + Storage layout

- The **width** of a **type** is the number of **storage units** needed for objects of that type.
- A **basic type**, such as a character, integer, or float, requires an integral number of bytes.
- **Arrays** allocated in one **contiguous block of bytes**

$T \rightarrow B \; C$	$\{\ t = B.type;\ w = B.width;\ \}$
$B \rightarrow \textbf{int}$	$\{\ B.type = integer;\ B.width = 4;\ \}$
$B \rightarrow \textbf{float}$	$\{\ B.type = float;\ B.width = 8;\ \}$
$C \rightarrow \epsilon$	$\{\ C.type = t;\ C.width = w;\ \}$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$\{\ array(\textbf{num}.value,\ C_1.type);$ $C.width = \textbf{num}.value \times C_1.width;\ \}$

Computes **data types** and their **widths** for basic and array types

Type Expressions

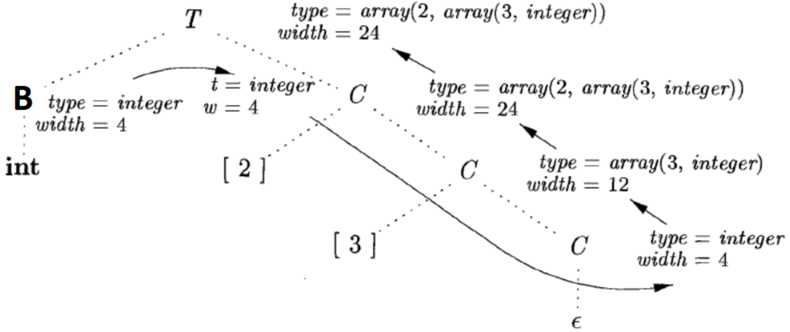
The **width of an array** is obtained by **multiplying** the **width of an element** by the **number of elements** in the array.

More on Declaration statement

Data type + Storage layout

```
int[2][3]
```

$T \rightarrow B$ $\quad\;\; C$		$\{\ t = B.type;\ w = B.width;\ \}$
$B \rightarrow$ **int**		$\{\ B.type = integer;\ B.width = 4;\ \}$
$B \rightarrow$ **float**		$\{\ B.type = float;\ B.width = 8;\ \}$
$C \rightarrow \epsilon$		$\{\ C.type = t;\ C.width = w;\ \}$
$C \rightarrow$ [**num**] C_1		$\{\ array(\textbf{num}.value,\ C_1.type);$ $C.width = \textbf{num}.value \times C_1.width;\ \}$



Relative address

Name	Data type
d	float
i	int
w	*array*(10, int)

Relative address
0
8
12

Sequences of Declarations

```
int x;
float y;
```

- Relative address: **offset**
- Keeps track of the **next available relative address**

$$
\begin{array}{lll}
P & \rightarrow & \{\ offset = 0;\ \}\ \ D \\
D & \rightarrow & T\ \mathbf{id}\ ; \quad \{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset); \\
 & & \qquad\qquad offset\ =\ offset + T.width;\ \} \\
 & & D_1 \\
D & \rightarrow & \epsilon
\end{array}
$$

- The translation scheme deals with a **sequence of declarations** of the form T id, where T generates a data type
- Before the first declaration is considered, **offset is set to 0**.
- As **each new name x** is seen, x is entered into the **symbol table** with its **relative address = current value of offset**,
  - which is **then incremented** by the width of the type of x.

Sequences of Declarations

- Relative address: offset
- Keeps track of the next available relative address

$$
\begin{array}{lll}
P & \rightarrow & \{\ \mathit{offset} = 0;\ \}\ \ D \\
D & \rightarrow & T\ \mathbf{id}\ ; \quad \{\ \mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset}); \\
 & & \qquad\qquad \mathit{offset}\ =\ \mathit{offset} + T.\mathit{width};\ \} \\
 & & D_1 \\
D & \rightarrow & \epsilon
\end{array}
$$

The semantic action within the production $D \rightarrow T\ \mathbf{id}\ ;\ D_1$ creates a symbol-table entry by executing $\mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset})$. Here *top* denotes the current symbol table. The method *top.put* creates a symbol-table entry for $\mathbf{id}.\mathit{lexeme}$, with type $T.\mathit{type}$ and relative address *offset* in its data area.

Record or Structure data type

$$T \rightarrow \textbf{record} \; '\{' \; D \; '\}'$$

The fields in this record type are specified by the sequence of declarations generated by D. The approach of Fig. 6.17 can be used to determine the types and relative addresses of fields, provided we are careful about two things:

- The field names within a record must be distinct; that is, a name may appear at most once in the declarations generated by D.
- The offset or relative address for a field name is relative to the data area for that record.

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

```
x = p.x + q.x;
```

Record or Structure data type

$$T \rightarrow \textbf{record} \text{ '\{' } D \text{ '\}'}$$

$$\begin{array}{llll} T & \rightarrow & \textbf{record} \text{ '\{'} & \{ \; Env.push(top); \; top = \textbf{new} \; Env(); \\ & & & \;\; Stack.push(offset); \; offset = 0; \; \} \\ & & D \text{ '\}'} & \{ \; T.type = record(top); \; T.width = offset; \\ & & & \;\; top = Env.pop(); \; offset = Stack.pop(); \; \} \end{array}$$

For convenience, record types will encode both the types and relative addresses of their fields, using a symbol table for the record type. A record type has the form *record*(t), where *record* is the type constructor, and t is a symbol-table object that holds information about the fields of this record type.

The embedded action before D saves the existing symbol table, denoted by *top* and sets *top* to a fresh symbol table. It also saves the current *offset*, and sets *offset* to 0. The declarations generated by D will result in types and relative addresses being put in the fresh symbol table. The action after D creates a record type using *top*, before restoring the saved symbol table and offset.

Record or Structure data type

$$T \rightarrow \textbf{record}\ '\{'\ D\ '\}'$$

$$\begin{array}{lll} T \rightarrow & \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env(); \\ & & \quad Stack.push(offset);\ offset = 0;\ \} \\ & D\ '\}' & \{\ T.type = record(top);\ T.width = offset; \\ & & \quad top = Env.pop();\ offset = Stack.pop();\ \} \end{array}$$

Let class *Env* implement symbol tables. The call *Env.push(top)* pushes the current symbol table denoted by *top* onto a stack. Variable *top* is then set to a new symbol table. Similarly, *offset* is pushed onto a stack called *Stack*. Variable *offset* is then set to 0.

After the declarations in D have been translated, the symbol table *top* holds the types and relative addresses of the fields in this record. Further, *offset* gives the storage needed for all the fields. The second action sets $T.type$ to *record(top)* and $T.width$ to *offset*. Variables *top* and *offset* are then restored to their pushed values to complete the translation of this record type.

Homework

```
int x, y;
float p, q
record
{ int x;
  float q;
} a;
char b;
```

Write the grammar, SDD and populate the symbol table by executing those rules

Major translation classes of Three address code generation

(a) Declaration statements (+ handling data type and storage)

(b) Expressions and statements

(a) Control flow statements

Translation of Expressions

statement a = b + - c

Three-address code for an assignment statement

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \ \|\|\ E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\quad\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \ \|\|$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\quad\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\quad\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

```
t1 = minus c
t2 = b + t1
a = t2
```

Attribute **code** for S

attributes **addr and code** for an expression E.

Attributes **S.code and E.code denote the three-address code** for S and E, respectively.

Attribute **E.addr denotes the address** that will hold the **value of E.**

Translation of Expressions

Three-address code for an assignment statement

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \;\|\|$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \;\|\|\; E_2.code \;\|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \;\|\|$ $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

When an expression is a **single identifier**, say x, then **x itself holds the value of the expression**.

The semantic rules for this production define **E.addr to point to the symbol-table entry**

Address of the variable

Translation of Expressions

Three-address code for an assignment statement

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \;\|\|$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \;\|\|\; E_2.code \;\|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$

The semantic rules for $E \rightarrow E_1 + E_2$, generate code to compute the value of E from the values of E_1 and E_2. Values are computed into newly generated temporary names. If E_1 is computed into $E_1.addr$ and E_2 into $E_2.addr$, then $E_1 + E_2$ translates into $t = E_1.addr + E_2.addr$, where t is a new temporary name. $E.addr$ is set to t. A sequence of distinct temporary names $t_1, t_2, \ldots$ is created by successively executing **new** $Temp()$.

Translation of Expressions

Three-address code for an assignment statement

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \ \|\|\ E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$

Finally, the production $S \rightarrow \textbf{id} = E;$ generates instructions that assign the value of expression E to the identifier **id**. The semantic rule for this production uses function *top.get* to determine the address of the identifier represented by **id**, as in the rules for $E \rightarrow \textbf{id}$. $S.code$ consists of the instructions to compute the value of E into an address given by $E.addr$, followed by an assignment to the address $top.get(\textbf{id}.lexeme)$ for this instance of **id**.

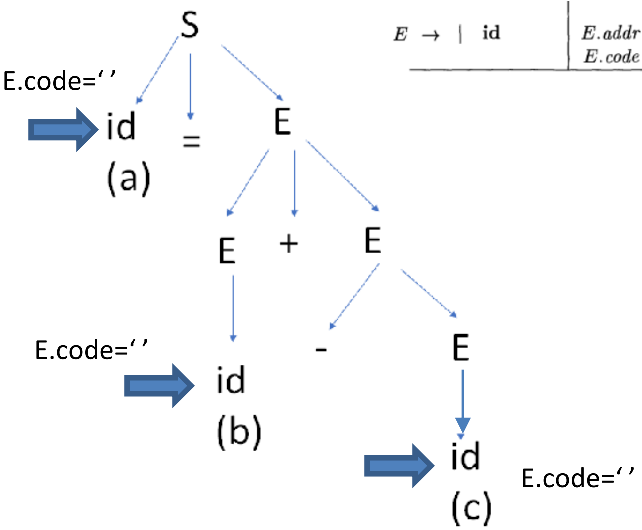
Translation of Expressions

Three-address code for an assignment statement

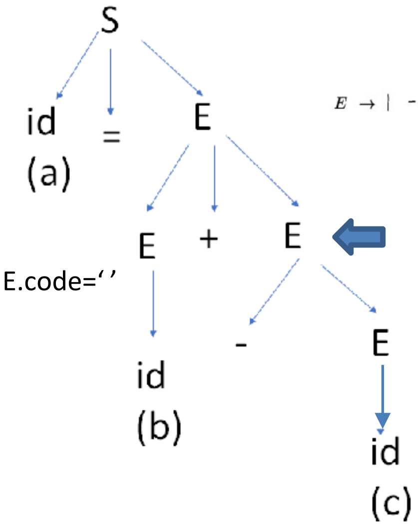
PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \ \|\|\ E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\quad \mid\ -E_1$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \ \|\|$ $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\quad \mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\quad \mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

statement a = b + − c

$E \rightarrow$	**id**	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$



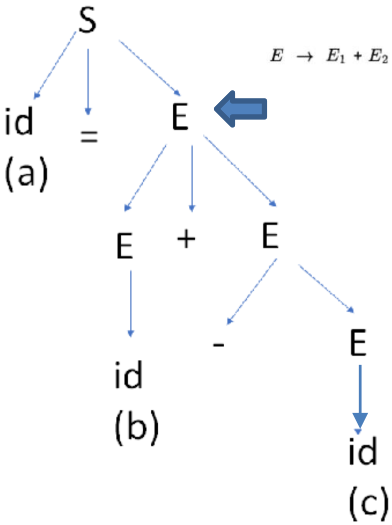
statement a = b + − c



$E \rightarrow - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \ \Vert$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$

E.code=

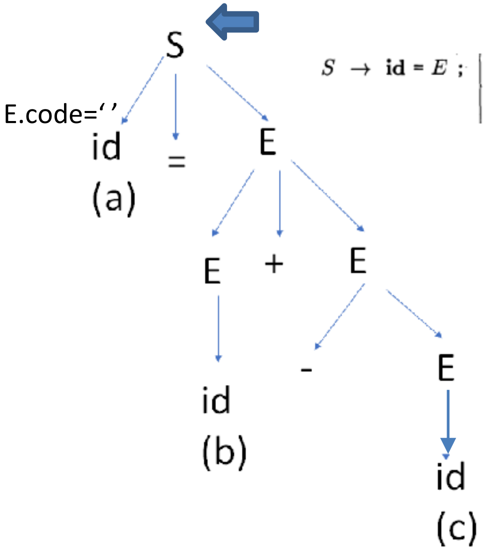
t_1 = minus c



$$E \rightarrow E_1 + E_2 \quad \Big| \quad \begin{array}{l} E.addr = \textbf{new}\ Temp\,() \\ E.code = E_1.code \;||\; E_2.code \;|| \\ \qquad\qquad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr) \end{array}$$

E.code=

```
t1 = minus c
t2 = b + t1
```



$$S \rightarrow \mathbf{id} = E \; ; \quad \Big| \quad S.code = E.code \; || \; gen(top.get(\mathbf{id}.lexeme) \; '=' \; E.addr)$$

S.code=

```
t1 = minus c
t2 = b + t1
a = t2
```

Translation of Expressions

Three-address code for an assignment statement

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \ \|\| \ E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \ \|\|$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

statement a = b + − c

```
t1 = minus c
t2 = b + t1
a = t2
```

Incremental Translation

- So far, ***E.Code*** attributes were long strings
  - Generated incrementally
- In incremental translation, generate only the **new three-address instructions**
- **Past sequence** may either be **retained in memory** for further processing, or it may be **output incrementally**.
- In the incremental approach, gen() not only constructs a three-address instruction,
  - it **appends** the instruction to the sequence of instructions generated so far.

Incremental Translation

$$
\begin{array}{lll}
S \rightarrow \textbf{id} = E \; ; & \{ \; gen(\; top.get(\textbf{id}.lexeme) \; '=' \; E.addr); \; \} \\
E \rightarrow E_1 + E_2 & \{ \; E.addr = \textbf{new} \; Temp(); \\
 & \;\; gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \; \} \\
\quad | \;\; - E_1 & \{ \; E.addr = \textbf{new} \; Temp(); \\
 & \;\; gen(E.addr \; '=' \; '\textbf{minus}' \; E_1.addr); \; \} \\
\quad | \;\; (\; E_1 \;) & \{ \; E.addr = E_1.addr; \; \} \\
\quad | \;\; \textbf{id} & \{ \; E.addr = top.get(\textbf{id}.lexeme); \; \}
\end{array}
$$

- This **translation scheme generates the same code** as the previous syntax directed definition.
- With the incremental approach, the **E.code attribute is not used,**
    - Since there is a **single sequence of instructions** that is created by **successive calls to gen()**.

Translation of Array Expressions

```
int num[3][4] = {
  {1, 2,  3,  4},
  {5, 6,  7,  8},
  {9, 10, 11, 12}
};
```

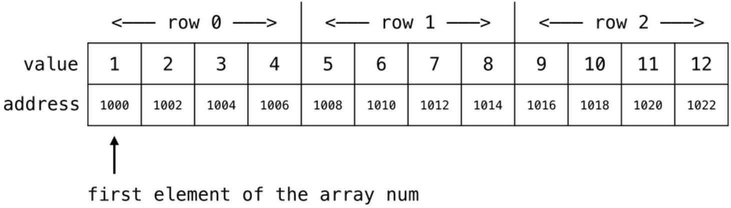
8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$. The instruction $x = y[i]$ sets x to the value in the location i memory units beyond location y. The instruction $x[i] = y$ sets the contents of the location i units beyond x to the value of y.

Translation of Array Expressions

```
int num[3][4] = {
  {1, 2,  3,  4},
  {5, 6,  7,  8},
  {9, 10, 11, 12}
};
```

Translate 2D index to 1D index

row-wise memory allocation



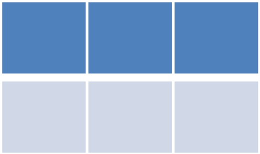
Let w1 be the length of a row (# of columns) and w2 be the size of an element in a row.
The relative address of A[i1][i2] can be calculated by the formula

$$base + i_1 \times w_1 + i_2 \times w_2$$

Translation of Array Expressions

Objective

a is a 2×3 array



expression c + a[i][j]

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

Translation of Array Expressions

$S \rightarrow \mathbf{id} = E\ ;$ $\{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \}$

$\mid\ L = E\ ;$ $\{\ gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \}$

$E \rightarrow E_1 + E_2$ $\{\ E.addr = \mathbf{new}\ Temp();$
$gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \}$

$\mid\ \mathbf{id}$ $\{\ E.addr = top.get(\mathbf{id}.lexeme);\ \}$

$\mid\ L$ $\{\ E.addr = \mathbf{new}\ Temp();$
$gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \}$

$L \rightarrow \mathbf{id}\ [\ E\ ]$ $\{\ L.array = top.get(\mathbf{id}.lexeme);$
$L.type = L.array.type.elem;$
$L.addr = \mathbf{new}\ Temp();$
$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \}$

$\mid\ L_1\ [\ E\ ]$ $\{\ L.array = L_1.array;$
$L.type = L_1.type.elem;$
$t = \mathbf{new}\ Temp();$
$L.addr = \mathbf{new}\ Temp();$
$gen(t\ '='\ E.addr\ '*'\ L.type.width);\ \}$
$gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}$

L.addr denotes a temporary variable containing the **offset** of the **array elements** (array index)

$$base + i_1 \times w_1 + i_2 \times w_2$$

Translation of Array Expressions

$$
\begin{array}{lll}
S & \rightarrow \textbf{id} = E\ ; & \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} \\
 & \mid\ L = E\ ; & \{\ gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} \\
E & \rightarrow E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
 & \mid\ \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \} \\
 & \mid\ L & \{\ E.addr = \textbf{new}\ Temp();\ gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} \\
L & \rightarrow \textbf{id}\ [\ E\ ] & \{\ L.array = top.get(\textbf{id}.lexeme);\ L.type = L.array.type.elem;\ L.addr = \textbf{new}\ Temp();\ gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
 & \mid\ L_1\ [\ E\ ] & \{\ L.array = L_1.array;\ L.type = L_1.type.elem;\ t = \textbf{new}\ Temp();\ L.addr = \textbf{new}\ Temp();\ gen(t\ '='\ E.addr\ '*'\ L.type.width);\ \}\ gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

- **L.array** is a pointer to the symbol-table entry for the array name.
- **L.array.base** indicates base address of the array -- array name
- **L.type** is the type t of the subarray generated by L.
- For **array type t**, we assume that its width is given by **t.width**.
- For any **array type t**, **t.elem** gives the type of array element.

Translation of Array Expressions

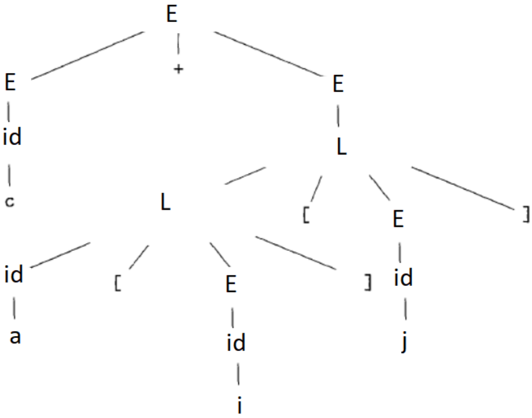
$$
\begin{array}{lll}
S \rightarrow \textbf{id} = E\ ; & \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} & \textbf{Standard} \\
\quad\mid\ L = E\ ; & \{\ gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} & \\
E \rightarrow E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp();\ & \\
 & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} & \\
\quad\mid\ \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \} & \textbf{Standard} \\
\quad\mid\ L & \{\ E.addr = \textbf{new}\ Temp(); & \\
 & \quad gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} & \\
L \rightarrow \textbf{id}\ [\ E\ ] & \{\ L.array = top.get(\textbf{id}.lexeme); & \\
 & \quad L.type = L.array.type.elem; & \\
 & \quad L.addr = \textbf{new}\ Temp(); & \\
 & \quad gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} & \\
\quad\mid\ L_1\ [\ E\ ] & \{\ L.array = L_1.array; & \\
 & \quad L.type = L_1.type.elem; & \\
 & \quad t = \textbf{new}\ Temp(); & \\
 & \quad L.addr = \textbf{new}\ Temp(); & \\
 & \quad gen(t\ '='\ E.addr\ '*'\ L.type.width);\ \} & \\
 & \quad gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \} &
\end{array}
$$

Translation of Array Expressions

$S \rightarrow$	**id** = E ;	{ $gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);$ }
$\mid$	L = E ;	{ $gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);$ }
$E \rightarrow$	$E_1 + E_2$	{ $E.addr = \mathbf{new}\ Temp();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }
$\mid$	**id**	{ $E.addr = top.get(\mathbf{id}.lexeme);$ }
$\mid$	L	{ $E.addr = \mathbf{new}\ Temp();$ $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }
$L \rightarrow$	**id** [E]	{ $L.array = top.get(\mathbf{id}.lexeme);$ $L.type = L.array.type.elem;$ $L.addr = \mathbf{new}\ Temp();$ $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);$ }
$\mid$	L_1 [E]	{ $L.array = L_1.array;$ $L.type = L_1.type.elem;$ $t = \mathbf{new}\ Temp();$ $L.addr = \mathbf{new}\ Temp();$ $gen(t\ '='\ E.addr\ '*'\ L.type.width);$ } $gen(L.addr\ '='\ L_1.addr\ '+'\ t);$ }

Translation of Array Expressions

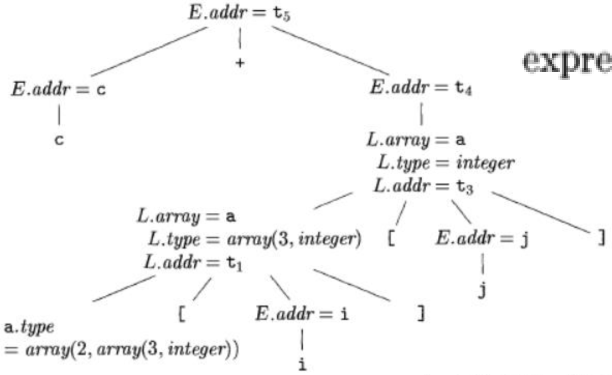
expression `c + a[i][j]`



Parse tree

$$S \rightarrow \mathbf{id} = E \;;$$
$$\quad |\;\; L = E \;;$$
$$E \rightarrow E_1 + E_2$$
$$\quad |\;\; \mathbf{id}$$
$$\quad |\;\; L$$
$$L \rightarrow \mathbf{id}\ [\ E\ ]$$
$$\quad |\;\; L_1\ [\ E\ ]$$

Translation of Array Expressions



expression c + a[i][j]

$E \rightarrow L$ { $E.addr$ = **new** $Temp()$;
$gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }

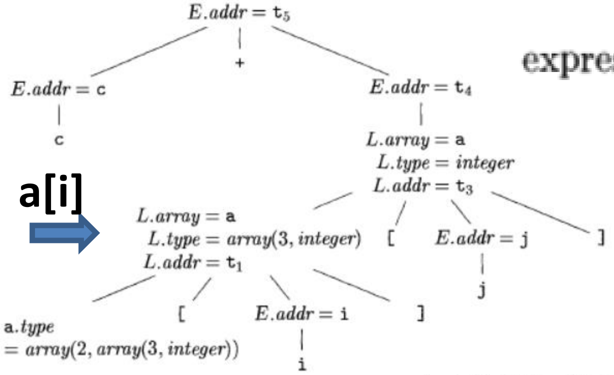
$E \rightarrow$ **id**   { $E.addr = top.get(\textbf{id}.lexeme)$; }

$L \rightarrow$ **id** [E]   { $L.array = top.get(\textbf{id}.lexeme)$;
$L.type = L.array.type.elem$;
$L.addr$ = **new** $Temp()$;
$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$; }

| L_1 [E]   { $L.array = L_1.array$;
$L.type = L_1.type.elem$;
t = **new** $Temp()$;
$L.addr$ = **new** $Temp()$;
$gen(t\ '='\ E.addr\ '*'\ L.type.width)$; }
$gen(L.addr\ '='\ L_1.addr\ '+'\ t)$; }

Translation of Array Expressions

expression c + a[i][j]



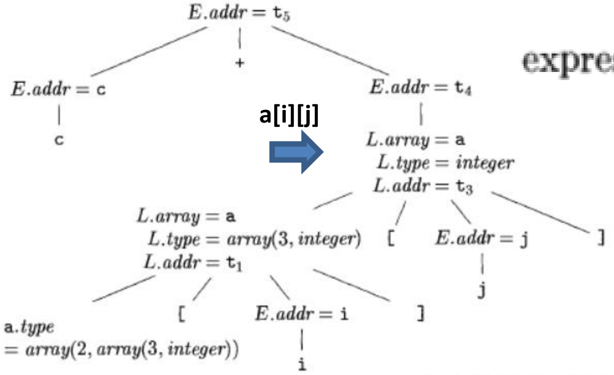
t_1 = i * 12

$L \rightarrow$ **id** [E] { $L.array = top.get($**id**$.lexeme)$;
$L.type = L.array.type.elem$;
$L.addr =$ **new** $Temp\,()$;
$gen(L.addr$ '=' $E.addr$ '*' $L.type.width)$; }

| L_1 [E] { $L.array = L_1.array$;
$L.type = L_1.type.elem$;
$t =$ **new** $Temp\,()$;
$L.addr =$ **new** $Temp\,()$;
$gen(t$ '=' $E.addr$ '*' $L.type.width)$; }
$gen(L.addr$ '=' $L_1.addr$ '+' $t)$; }

$E \rightarrow$ **id** { $E.addr = top.get($**id**$.lexeme)$; }

Translation of Array Expressions



expression c + a[i][j]

$$t_1 = i * 12$$

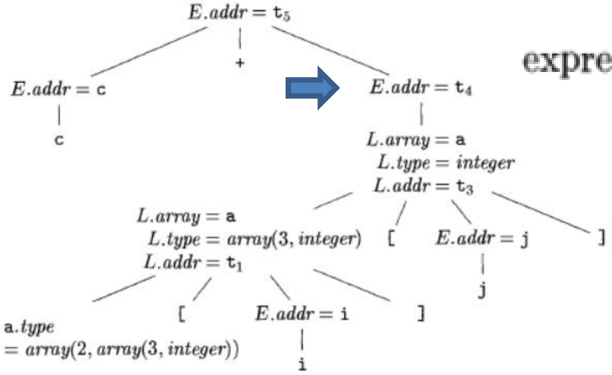
$$t_2 = j * 4$$
$$t_3 = t_1 + t_2$$

$L \rightarrow$ **id** [E] { $L.array = top.get($**id**$.lexeme)$; $L.type = L.array.type.elem$; $L.addr =$ **new** $Temp\,()$; $gen(L.addr$ '=' $E.addr$ '*' $L.type.width)$; }

| L_1 [E] { $L.array = L_1.array$; $L.type = L_1.type.elem$; $t =$ **new** $Temp\,()$; $L.addr =$ **new** $Temp\,()$; $gen(t$ '=' $E.addr$ '*' $L.type.width)$; $gen(L.addr$ '=' $L_1.addr$ '+' $t)$; }

$E \rightarrow$ **id** { $E.addr = top.get($**id**$.lexeme)$; }

Translation of Array Expressions



expression c + a[i][j]

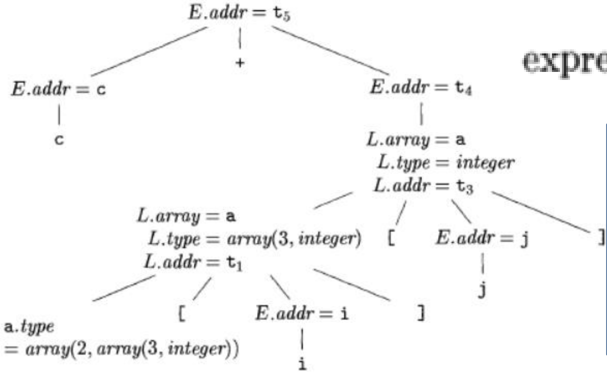
t_1 = i * 12

t_2 = j * 4
t_3 = t_1 + t_2

t_4 = a [t_3]

$E \rightarrow L \qquad \{\ E.addr = \textbf{new}\ Temp();$
$\qquad gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \}$

Translation of Array Expressions



expression c + a[i][j]

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

$L \rightarrow$ **id** [E] { $L.array = top.get(\textbf{id}.lexeme)$;
$L.type = L.array.type.elem$;
$L.addr =$ **new** $Temp()$;
$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$; }

| L_1 [E] { $L.array = L_1.array$;
$L.type = L_1.type.elem$;
$t =$ **new** $Temp()$;
$L.addr =$ **new** $Temp()$;
$gen(t\ '='\ E.addr\ '*'\ L.type.width)$; }
$gen(L.addr\ '='\ L_1.addr\ '+'\ t)$; }

$E \rightarrow L$ { $E.addr =$ **new** $Temp()$;
$gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }

$E \rightarrow$ **id** { $E.addr = top.get(\textbf{id}.lexeme)$; }

Translation of Array Expressions

$$
\begin{array}{llll}
S & \rightarrow & \textbf{id} = E \; ; & \{\; gen(\; top.get(\textbf{id}.lexeme) \; '=' \; E.addr); \; \} \\
 & | & L = E \; ; & \{\; gen(L.addr.base \; '[' \; L.addr \; ']' \; '=' \; E.addr); \; \} \\
E & \rightarrow & E_1 + E_2 & \{\; E.addr = \textbf{new}\; Temp(); \\
 & & & \quad gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \; \} \\
 & | & \textbf{id} & \{\; E.addr = top.get(\textbf{id}.lexeme); \; \} \\
 & | & L & \{\; E.addr = \textbf{new}\; Temp(); \\
 & & & \quad gen(E.addr \; '=' \; L.array.base \; '[' \; L.addr \; ']'); \; \} \\
L & \rightarrow & \textbf{id} \; [\; E \;] & \{\; L.array = top.get(\textbf{id}.lexeme); \\
 & & & \quad L.type = L.array.type.elem; \\
 & & & \quad L.addr = \textbf{new}\; Temp(); \\
 & & & \quad gen(L.addr \; '=' \; E.addr \; '*' \; L.type.width); \; \} \\
 & | & L_1 \; [\; E \;] & \{\; L.array = L_1.array; \\
 & & & \quad L.type = L_1.type.elem; \\
 & & & \quad t = \textbf{new}\; Temp(); \\
 & & & \quad L.addr = \textbf{new}\; Temp(); \\
 & & & \quad gen(t \; '=' \; E.addr \; '*' \; L.type.width); \; \} \\
 & & & \quad gen(L.addr \; '=' \; L_1.addr \; '+' \; t); \; \}
\end{array}
$$

L.addr denotes a temporary variable containing the **offset** of the **array elements** (array index)

$$base + i_1 \times w_1 + i_2 \times w_2$$

Type Conversions

- Consider expressions like x + i, where x is of type float and i is of type integer
- Since the representation of integers and floating-point numbers is different within a computer
- Compiler may need to convert one of the operands of + to ensure that both operands are of the same type when the addition occurs.

Suppose that integers are converted to floats when necessary, using a unary operator (`float`). For example, the integer 2 is converted to a float in the code for the expression `2 * 3.14`:

$$t_1 = \texttt{(float) 2}$$
$$t_2 = t_1 \; * \; 3.14$$

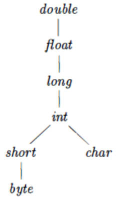
Type Conversions

We introduce another attribute $E.type$, whose value is either *integer* or *float*. The rule associated with $E \rightarrow E_1 + E_2$ builds on the pseudocode

$$\textbf{if } (E_1.type = integer \textbf{ and } E_2.type = integer) \; E.type = integer;$$
$$\textbf{else if } (E_1.type = float \textbf{ and } E_2.type = integer) \; \cdots$$
$$\cdots$$

Type conversion rules

widening conversions, which are intended to preserve information,



- Conversion from one type to another is said to be **implicit** if it is done **automatically by the compiler**.
- Conversion is said to be **explicit** if the programmer must write something to cause the conversion.
    - Explicit conversions are also called **type casts**

The semantic action for checking $E \rightarrow E_1 + E_2$ uses two functions:

1. $max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t_1 or t_2 is not in the hierarchy; e.g., if either type is an array or a pointer type.

2. $widen(a, t, w)$ generates type conversions if needed to widen an address a of type t into a value of type w. It returns a itself if t and w are the same type. Otherwise, it generates an instruction to do the conversion and place the result in a temporary t, which is returned as the result. Pseudocode for *widen*, assuming that the only types are *integer* and *float*,

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

$$
\begin{array}{llll}
S & \rightarrow & \textbf{id} = E\ ; & \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} \\
E & \rightarrow & E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp();\\
 & & & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
 & | & -\ E_1 & \{\ E.addr = \textbf{new}\ Temp();\\
 & & & \quad gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr);\ \} \\
 & | & (\ E_1\ ) & \{\ E.addr = E_1.addr;\ \} \\
 & | & \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \}
\end{array}
$$

New semantic action illustrates how type conversions can be added to the SDT for translating expressions

$$
\begin{aligned}
S &\rightarrow \textbf{id} = E \; ; && \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} \\
E &\rightarrow E_1 + E_2 && \{\ E.addr = \textbf{new}\ Temp();\ \\
& && \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
&\mid\ -E_1 && \{\ E.addr = \textbf{new}\ Temp(); \\
& && \quad gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr);\ \} \\
&\mid\ (\ E_1\ ) && \{\ E.addr = E_1.addr;\ \} \\
&\mid\ \textbf{id} && \{\ E.addr = top.get(\textbf{id}.lexeme);\ \}
\end{aligned}
$$

New semantic action illustrates how type conversions can be added to the SDT for translating expressions

$$
\begin{aligned}
E \rightarrow E_1 + E_2 \quad \{\ & E.type \;=\; max(E_1.type, E_2.type); \\
& a_1 \;=\; widen(E_1.addr, E_1.type, E.type); \\
& a_2 \;=\; widen(E_2.addr, E_2.type, E.type); \\
& E.addr = \textbf{new}\ Temp(); \\
& gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{aligned}
$$

Major translation classes of Three address code generation

(a) Declaration statements (+ handling data type and storage)

(b) Expressions and statements

(a) Control flow statements

Control Flow

Translation of statements such **as if-else-statements and while-statements**

Key step: Translation of **boolean expressions**

$$
\begin{aligned}
S &\rightarrow \textbf{if} \ (\ B \ ) \ S_1 \\
S &\rightarrow \textbf{if} \ (\ B \ ) \ S_1 \ \textbf{else} \ S_2 \\
S &\rightarrow \textbf{while} \ (\ B \ ) \ S_1
\end{aligned}
$$

Boolean expressions are used as

(i) Conditional expressions in statements that alter the **flow of control**

(ii) A boolean expression can **evaluate true Or false as values**. Such boolean expressions can be evaluated in analogy to **arithmetic expressions** using three-address instructions with logical operators

- The **intended use of boolean expressions** is determined by its **syntactic context**.
- We concentrate on the use of boolean expressions to **alter the flow of control.**
    - For clarity, we introduce a **new nonterminal B**

Control Flow: Three address codes

4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (`<`, `==`, `>=`, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y. If not, the three-address instruction following `if` x *relop* y `goto` L is executed next, in sequence.

Boolean Expressions

$$B \rightarrow B \;||\; B \;|\; B \;\&\&\; B \;|\; !\, B \;|\; (\, B\,) \;|\; E \;\textbf{rel}\; E \;|\; \textbf{true} \;|\; \textbf{false}$$

$<, <=, =, !=, >,$ or $>=$ is represented by **rel**.

Given the expression $B_1 \;||\; B_2$, if we determine that B_1 is true, then we can conclude that the entire expression is true without having to evaluate B_2. Similarly, given $B_1 \&\& B_2$, if B_1 is false, then the entire expression is false.

The semantic definition of the programming language determines whether all parts of a boolean expression must be evaluated.

Short-Circuit Code

In *short-circuit* (or *jumping*) code, the boolean operators &&, ||, and ! translate into jumps.

The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.

Control Flow

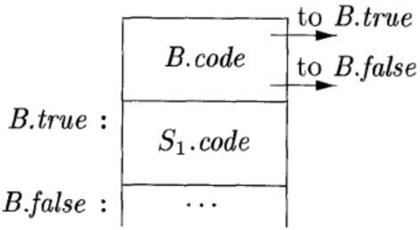
```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

```
     if x < 100 goto L2
     ifFalse x > 200 goto L1
     ifFalse x != y goto L1
L2:  x = 0
L1:
```

Translation of Flow-of-Control Statements into three-address code

$$
\begin{array}{lll}
S & \rightarrow & \textbf{if} \; (\; B \;) \; S_1 \\
S & \rightarrow & \textbf{if} \; (\; B \;) \; S_1 \; \textbf{else} \; S_2 \\
S & \rightarrow & \textbf{while} \; (\; B \;) \; S_1
\end{array}
$$

- **Nonterminal *B*** represents a boolean expression and **nonterminal S** represents a statement.
- Both **B and S** have a **synthesized attribute *code***, which gives the translation into three-address instructions.
- **Inherited** attributes **B.true, B.false, S.next** generate **labels** for control flow
- **B.true the label** to which control flows if B is true,
- **B.false, the label** to which control flows if B is false.
- With a statement S, we associate an inherited attribute **S.next** denoting **a label** for the instruction **immediately after the code for S**.



(a) if

$$S \rightarrow \textbf{if}\ (B)\ S_1$$

- **Nonterminal *B*** represents a boolean expression and **nonterminal S** represents a statement.
- Both **B and S** have a **synthesized attribute code**, which gives the translation into three-address instructions.
- Inherited attributes **B.true, B.false, S.next** generate labels for control flow
- **B.true the label** to which control flows if B is true,
- **B.false, the label** to which control flows if B is false.
- With a statement S, we associate an inherited attribute **S.next** denoting **a label** for the instruction **immediately after the code for S**.

Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

- **newlabel()** creates a new **label** each time it is called,
- **label(L)** attaches label L to the next three-address instruction to be generated

- A **program consists** of a statement generated by **P -> S**.
- The semantic rules associated with this production initialize **S.next** to a new label.
- **P.code** consists of **S.code** followed by the new label **S.next**.

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

⇦ standard

S→ id=E;

assign in the production *S* —> **assign** is a placeholder for assignment statements.

- In translating *S* **->** **if** *(B) S1,* the semantic rules create a **new label *B.true*** and **attach** it to the **first three-address instruction** generated for the statement *S1*.
- Thus, **jumps to *B.true*** within the code for *B* will go to the **code *S1***.
- By **setting *B.false* to *S.next***, we ensure that control will **skip the code for *S1*** if *B* evaluates to **false**.

Translation of Expressions

statement a = b + - c

Three-address code for an assignment statement

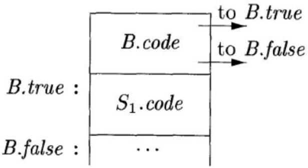
PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \parallel gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \parallel E_2.code \parallel gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \parallel gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

t_1 = minus c
t_2 = b + t_1
a = t_2

Attribute **code** for S
attributes **addr and code** for an expression E.
Attributes **S.code and E.code denote the three-address code** for S and E, respectively.
Attribute **E.addr denotes the address** that will hold the **value of E.**

$$S \rightarrow \textbf{if} \; (\; B \;) \; S_1$$

$$\begin{aligned} B.true &= newlabel() \\ B.false &= S_1.next = S.next \\ S.code &= B.code \;||\; label(B.true) \;||\; S_1.code \end{aligned}$$



(a) if

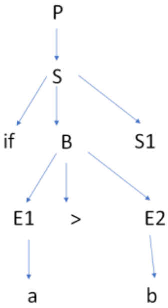
- In translating *S* **->** **if** *(B) S1,* the semantic rules create a **new label *B.true*** and **attach** it to the **first three-address instruction** generated for the statement *S1*.
- Thus, **jumps to *B.true*** within the code for *B* will go to the **code *S1*.**
- By **setting *B.false* to *S.next*,** we ensure that control will **skip the code for *S1*** if *B* evaluates to **false**.

Translation of Boolean Expressions

$$B \rightarrow E_1 \ \textbf{rel} \ E_2 \quad \Big| \quad B.code = E_1.code \ || \ E_2.code \ || \ gen(\text{'if'} \ E_1.addr \ \textbf{rel}.op \ E_2.addr \ \text{'goto'} \ B.true) \ || \ gen(\text{'goto'} \ B.false)$$

if a>b

x=0



$$P \rightarrow S$$

$$S \rightarrow \textbf{assign}$$ S→ id=E;

$$S \rightarrow \textbf{if} \ (\ B \ ) \ S_1$$

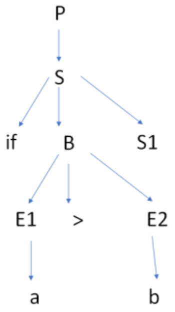
PRODUCTION	SEMANTIC RULES
$P \to S$	$S.next = newlabel()$ $P.code = S.code \ \|\| \ label(S.next)$
$S \to \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \to \textbf{if} \ (\ B \ ) \ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \ \|\| \ label(B.true) \ \|\| \ S_1.code$

S→ id=E;

S.next=L1

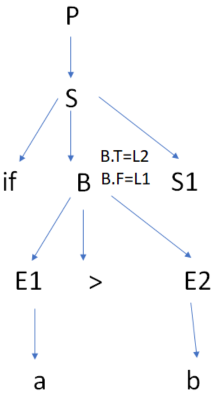
P.Code:

S.code
L1:



if a>b
x=0

Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \Vert label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \Vert label(B.true) \Vert S_1.code$



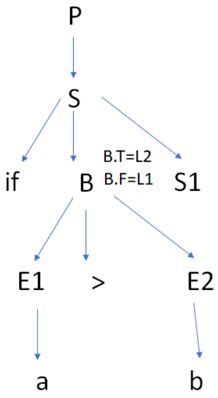
S.next=L1
B.true= L2
B. false=L1

P.Code:

B.code
L2:
S1.code
L1:

if a>b
x=0

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$



S.next=L1

B.true= L2

B. false=L1

P.Code:

if a>b goto L2

goto L1

L2: S1

L1:……

if a>b

x=0

$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}' \; E_1.addr \; \textbf{rel}.op \; E_2.addr \; '\texttt{goto}' \; B.true)$ $\mid\mid gen('\texttt{goto}' \; B.false)$

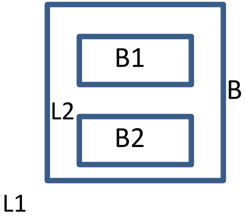
Translation of Boolean Expressions

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$

If B1 is true, then we immediately know that B itself is true, so B1.true is the same as B.true.

If B1 is false, then B2 must be evaluated, so B1.false gets the label of the first instruction in the code for B2

The true and false exits of B2 are the same as the true and false exits of B, respectively.



Translation of Boolean Expressions

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \, \&\& \, B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \, ! \, B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'goto'}\ B.true)$ $\mid\mid gen(\text{'goto'}\ B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen(\text{'goto'}\ B.true)$
$B \rightarrow \textbf{false}$	$B.code = gen(\text{'goto'}\ B.false)$

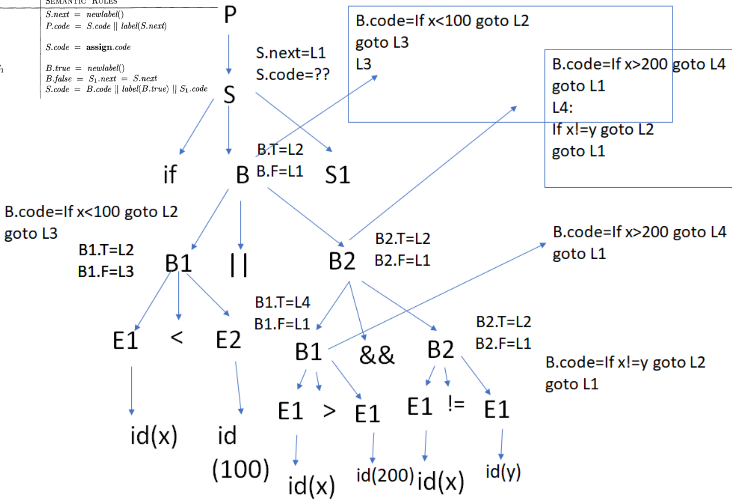
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$

$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

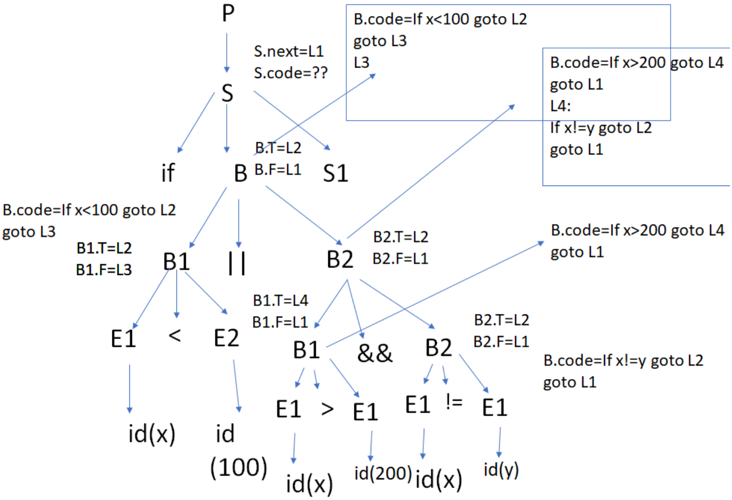
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \;\|\|\; E_2.code$ $\|\|\; gen('\texttt{if}'\; E_1.addr\; \textbf{rel}.op\; E_2.addr\; '\texttt{goto}'\; B.true)$ $\|\|\; gen('\texttt{goto}'\; B.false)$

Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**.$code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\|\; label(B.true) \;\|\|\; S_1.code$



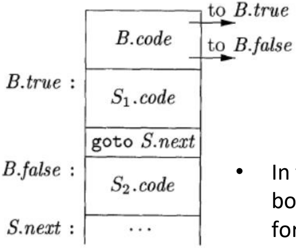
```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

```
if( x < 100 || x > 200 && x != y ) x = 0;
```



Translation of if-else statement

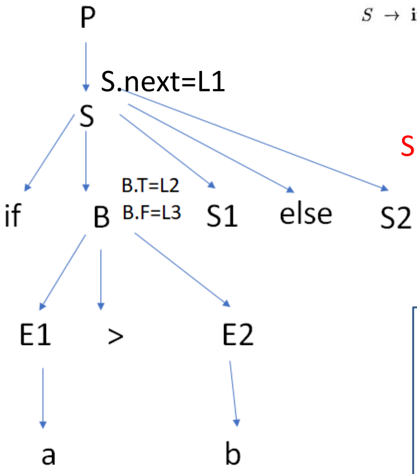
PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \; \vert\vert \; label(S.next)$
$S \rightarrow \textbf{if} (B) S_1 \textbf{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\vert\vert \; label(B.true) \; \vert\vert \; S_1.code$ $\vert\vert \; gen(\text{'goto'} \; S.next)$ $\vert\vert \; label(B.false) \; \vert\vert \; S_2.code$



(b) if-else

- In translating the if-else-statement, **if B is true** , the code for the boolean expression B has **jumps to the first instruction** of the code for **S1**,
- if **B is false**, control jumps to the **first instruction of the code for S2**.
- Further, control flows **from both S1 and S2** to the three-address instruction **immediately following the code for S** — its label is given by the inherited attribute **S.next.**
- An explicit **goto S.next** appears after the code for **S1 to skip over** the code for S2 . **No goto is needed after S2** , since S2.next is the same as S.next.

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{if} (B) S_1 \textbf{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('goto'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$



S→ id=E;

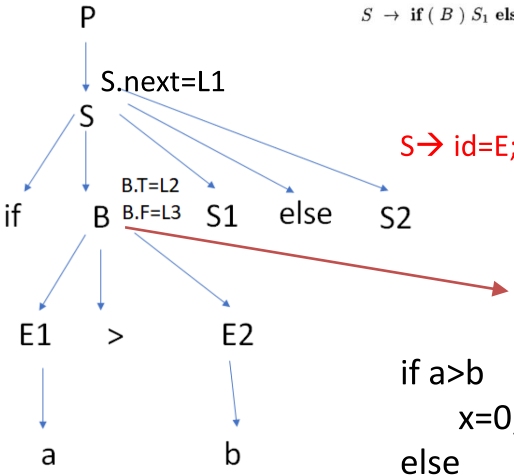
```
if a>b
      x=0;
else
      y=0;
```

L1:

$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('if'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ 'goto'\ B.true)$ $\mid\mid gen('goto'\ B.false)$

Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \ \|\| \ label(S.next)$
$S \rightarrow \textbf{if} \ (\ B \ ) \ S_1 \ \textbf{else} \ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\quad \|\| \ label(B.true) \ \|\| \ S_1.code$ $\quad \|\| \ gen('\texttt{goto}' \ S.next)$ $\quad \|\| \ label(B.false) \ \|\| \ S_2.code$

S→ id=E;



B.code=

if a>b goto L2

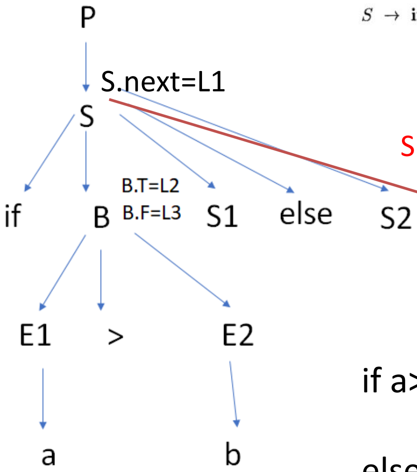
goto L3

```
if a>b
    x=0;
else
    y=0;
```

L1:

$B \rightarrow E_1 \ \textbf{rel} \ E_2$	$B.code = E_1.code \ \|\| \ E_2.code$ $\|\| \ gen('\texttt{if}' \ E_1.addr \ \textbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ $\|\| \ gen('\texttt{goto}' \ B.false)$

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{if} \ (\ B \ ) \ S_1 \ \textbf{else} \ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('goto' \ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$



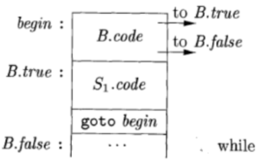
S→ id=E;

```
if a>b
      x=0;
else
      y=0;
```

S.code=

```
if a>b goto L2
goto L3
L2: x=0
    goto L1
L3: y=0
L1:
```

Translation of while statement



$$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$$

$$\begin{aligned}
begin &= newlabel() \\
B.true &= newlabel() \\
B.false &= S.next \\
S_1.next &= begin \\
S.code &= label(begin)\ ||\ B.code \\
&\quad ||\ label(B.true)\ ||\ S_1.code \\
&\quad ||\ gen(\texttt{'goto'}\ begin)
\end{aligned}$$

- We use a **local variable** **begin** to hold a new label attached to the **first instruction** for this while-statement,
  - which is also the first instruction for B.
  - Variable begin is local to the semantic rules for this production.
- The **inherited label S.next** marks the instruction that control must flow to **if B is false**; hence, **B.false is set to be S.next**.
- A **new label B.true** is **attached** to the **first instruction for S1**;
  - the code for B generates a jump to this label if B is true.
- **After the code for S1** we place the instruction **goto begin**, which causes a **jump back** to the **beginning of the code** for the boolean expression.
- Note that S1.next is set to this label begin

Translation of while statement

Homework

```
while(a>b)
 x=0;
```

Avoiding Redundant Gotos

```
        if x > 200 goto L4
        goto L1
L4 :    ...
```

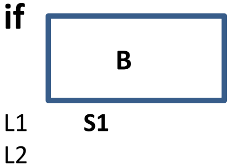
Free fall to L4 when x>200

Jumps to L1 only when x<=200

Instead, consider the instruction:

```
      ifFalse x > 200 goto L1
L4 :  ...
```

The code for statement S1 immediately follows the code for the boolean expression B



$$S \rightarrow \textbf{if}\ (B)\ S_1$$

$$B.true = fall$$
$$B.false = S_1.next\ =\ S.next$$
$$S.code = B.code\ ||\ S_1.code$$

Jump when false

$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true\ =\ newlabel()$ **L1** $B.false\ =\ S_1.next\ =\ S.next$ $S.code\ =\ B.code\ \|\|\ label(B.true)\ \|\|\ S_1.code$

Avoiding Redundant Gotos

$$B \rightarrow E_1 \ \mathbf{rel} \ E_2$$

$$\begin{aligned}
&test = E_1.addr \ \mathbf{rel}.op \ E_2.addr \\
&s = \mathbf{if} \ B.true \neq fall \ \mathbf{and} \ B.false \neq fall \ \mathbf{then} \\
&\qquad gen('\mathtt{if}' \ test \ '\mathtt{goto}' \ B.true) \ || \ gen('\mathtt{goto}' \ B.false) \\
&\quad \mathbf{else\ if} \ B.true \neq fall \ \mathbf{then} \ gen('\mathtt{if}' \ test \ '\mathtt{goto}' \ B.true) \\
&\quad \mathbf{else\ if} \ B.false \neq fall \ \mathbf{then} \ gen('\mathtt{ifFalse}' \ test \ '\mathtt{goto}' \ B.false) \\
&\quad \mathbf{else} \ '' \\
&B.code = E_1.code \ || \ E_2.code \ || \ s
\end{aligned}$$

Jump when true

Jump when false

$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \ \|\| \ E_2.code$ $\|\| \ gen('\mathtt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\mathtt{goto}' \ B.true)$ $\|\| \ gen('\mathtt{goto}' \ B.false)$

Avoiding Redundant Gotos

```
if x>200
    a=0
```

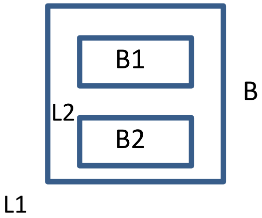
Avoiding Redundant Gotos

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \;\|\|\; B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$

Semantic rules for $B \rightarrow B_1 \;||\; B_2$

$$
\begin{aligned}
B_1.true &= \textbf{if } B.true \neq fall \textbf{ then } B.true \textbf{ else } newlabel() \\
B_1.false &= fall \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= \textbf{if } B.true \neq fall \textbf{ then } B_1.code \;||\; B_2.code \\
&\quad\; \textbf{else } B_1.code \;||\; B_2.code \;||\; label(B_1.true)
\end{aligned}
$$

- **Meaning** of label **fall for B** is different from its **meaning for B1**.
- Suppose **B.true is fall**; i.e, control **falls through B**, if B evaluates to true.
- Although B evaluates to true if B1 does, **B1.true must ensure that control jumps over the code for B2** to get to the next instruction after B.



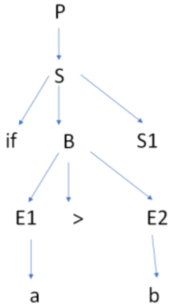
PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \ \|\| \ label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} \ (\ B \ ) \ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \ \|\| \ label(B.true) \ \|\| \ S_1.code$

$S \rightarrow id=E;$

S.next=L1

P.Code:

S.code
L1:



If x>100 || a>b
x=0;

Avoiding Redundant Gotos

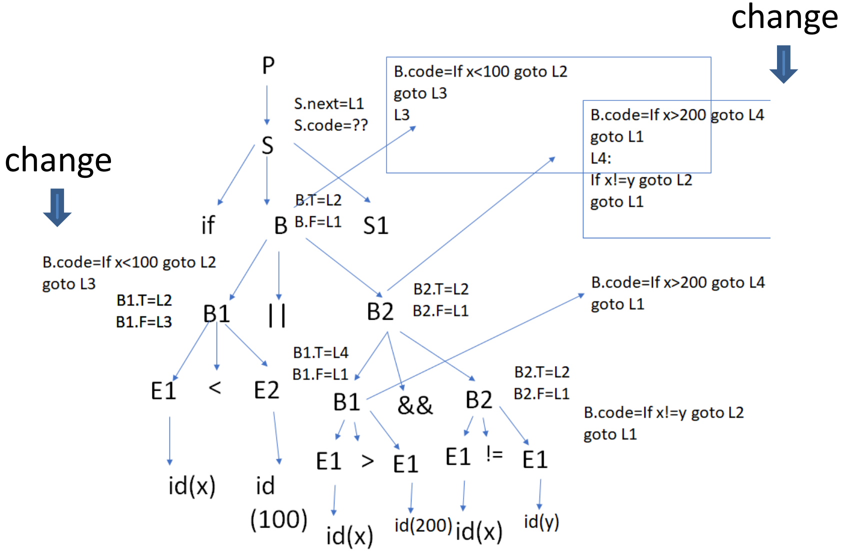
```
if( x < 100 || x > 200 && x != y ) x = 0;
```

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

Avoiding Redundant Gotos

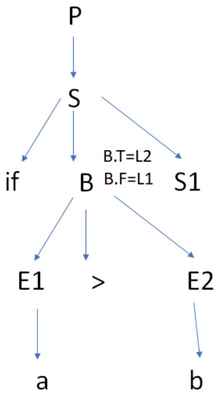
```
if( x < 100 || x > 200 && x != y ) x = 0;
```



	$\| gen('\text{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \,\|\, label(S_1.next) \,\|\, S_2.code$

Backpatching: Problem

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \ \|\| \ label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} \ (\ B \ ) \ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \ \|\| \ label(B.true) \ \|\| \ S_1.code$



S.next=L1
B.true= L2
B. false=L1

P.Code:

if a>b goto L2
goto L1
L2: S1
L1:……

if a>b
x=0

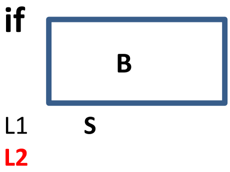
$B \rightarrow E_1 \ \textbf{rel} \ E_2$	$B.code = E_1.code \ \|\| \ E_2.code$ $\|\| \ gen('\texttt{if}' \ E_1.addr \ \textbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ $\|\| \ gen('\texttt{goto}' \ B.false)$

Backpatching

- **Key problem** when generating code for **boolean expressions and flow-of-control statements** is that
- Matching a **jump instruction** with the **target of the jump**.
- For example, the translation of the **boolean expression B in if (B) S** contains a **jump**,
  - when B is false, jump **to the instruction following the code for S**.

- In a one-pass translation, **B must be translated before S is examined**.

- What then is the **target of the goto that jumps** over the code for S?



$S \rightarrow \textbf{if} \; (\; B \;) \; S_1$

$$\begin{aligned} B.true &= newlabel() \\ B.false &= S_1.next \;=\; S.next \\ S.code &= B.code \;||\; label(B.true) \;||\; S_1.code \end{aligned}$$

Backpatching

- **Lists of jumps** are passed as **synthesized attributes**.
- When a **jump is generated**, the **target** of the jump is temporarily left **unspecified**.
- Each such **jump** is put on a **list of jumps** whose **labels are to be filled** in **when the proper label** can be determined.
- **All of the jumps** on a list have the **same target label**.

- **Synthesized attributes truelist and falselist** of nonterminal **B** are used to manage labels.
- **B.truelist** will be a **list of jump or conditional jump instructions** into which, **we must insert the label** to which **control goes if B is true**.
- **B.falselist** likewise is the **list of instructions** that eventually get the **label** to which **control goes when B is false**.
- **Code is generated for B**, jumps to the true and false exits are left incomplete, with the **label field unfilled**.
- These **incomplete jumps** are placed on lists pointed to by **B.truelist and B.falselist**, as appropriate.

Backpatching

- We generate instructions into an **instruction array**,
- **Labels** will be **indices** into this array.
- To manipulate lists of jumps, we use **three functions:**

1. **makelist(i)** creates a new list containing an index i into the array of instructions; makelist returns a pointer to the newly created list.

2. **merge(p1,p2)** concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.

3. **backpatch(p,i)** inserts **i as the target label** for each of the instructions on the list pointed to by **p**.

Backpatching for Boolean Expressions

- We now construct a **translation scheme** suitable for generating code for **Boolean expressions** during **bottom-up parsing**.

$$B \rightarrow E_1 \ \mathbf{rel} \ E_2 \qquad \{ \ B.truelist = makelist(nextinstr); \ B.falselist = makelist(nextinstr + 1); \ emit('\mathtt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\mathtt{goto} \ \_'); \ emit('\mathtt{goto} \ \_'); \ \}$$

$$B \rightarrow \mathbf{true} \qquad \{ \ B.truelist = makelist(nextinstr); \ emit('\mathtt{goto} \ \_'); \ \}$$

$$B \rightarrow \mathbf{false} \qquad \{ \ B.falselist = makelist(nextinstr); \ emit('\mathtt{goto} \ \_'); \ \}$$

Semantic actions generates two instructions, **a conditional goto** and an **unconditional goto**.
Neither has its **target filled in**.
These **instructions** are put on **new lists**, pointed to by **B. truelist** and **B .falselist**, respectively

Backpatching for Boolean Expressions

- We now construct a **translation scheme** suitable for generating code for **Boolean expressions** during **bottom-up parsing**.
- A **marker nonterminal M** causes a semantic action to pick up, at appropriate times, the **index of the next instruction** to be generated.

1) $B \rightarrow B_1 \mid\mid M \; B_2$ $\quad \{ \; backpatch(B_1.falselist, M.instr);$
$\quad B.truelist = merge(B_1.truelist, B_2.truelist);$
$\quad B.falselist = B_2.falselist; \; \}$

- If B1 is true, then B is also true, so the jumps on **B1.truelist become part of B.truelist**.
- If **B1 is false**, however, we must next **test B2**,
- So the **target for the jumps B1.falselist** must be the **beginning** of the code generated for **B2**.
- **This target** is obtained using the **marker nonterminal M**.
- That nonterminal M produces, as a **synthesized attribute M.instr**, the index of the next instruction, **just before B2 code** starts being generated.

Backpatching for Boolean Expressions

To obtain that instruction index, we associate with the production $M \to \epsilon$ the semantic action

$$\{ \; M.instr = nextinstr; \; \}$$

The variable *nextinstr* holds the index of the next instruction to follow. This value will be backpatched onto the $B_1.falselist$ (i.e., each instruction on the list $B_1.falselist$ will receive $M.instr$ as its target label) when we have seen the remainder of the production $B \to B_1 \; || \; M \; B_2$.

1) $B \to B_1 \; || \; M \; B_2$ $\quad$ $\{ \; backpatch(B_1.falselist, M.instr);$
$B.truelist = merge(B_1.truelist, B_2.truelist);$
$B.falselist = B_2.falselist; \; \}$

2) $B \to B_1 \; \&\& \; M \; B_2$ $\quad$ $\{ \; backpatch(B_1.truelist, M.instr);$
$B.truelist = B_2.truelist;$
$B.falselist = merge(B_1.falselist, B_2.falselist); \; \}$

3) $B \to \; ! \; B_1$ $\quad$ $\{ \; B.truelist = B_1.falselist;$
$B.falselist = B_1.truelist; \; \}$

4) $B \to (\; B_1 \;)$ $\quad$ $\{ \; B.truelist = B_1.truelist;$
$B.falselist = B_1.falselist; \; \}$

5) $B \to E_1 \; \mathbf{rel} \; E_2$ $\quad$ $\{ \; B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$emit(\text{'}\texttt{if}\text{'} \; E_1.addr \; \mathbf{rel}.op \; E_2.addr \; \text{'}\texttt{goto \_}\text{'});$
$emit(\text{'}\texttt{goto \_}\text{'}); \; \}$

6) $B \to \mathbf{true}$ $\quad$ $\{ \; B.truelist = makelist(nextinstr);$
$emit(\text{'}\texttt{goto \_}\text{'}); \; \}$

7) $B \to \mathbf{false}$ $\quad$ $\{ \; B.falselist = makelist(nextinstr);$
$emit(\text{'}\texttt{goto \_}\text{'}); \; \}$

8) $M \to \epsilon$ $\quad$ $\{ \; M.instr = nextinstr; \; \}$

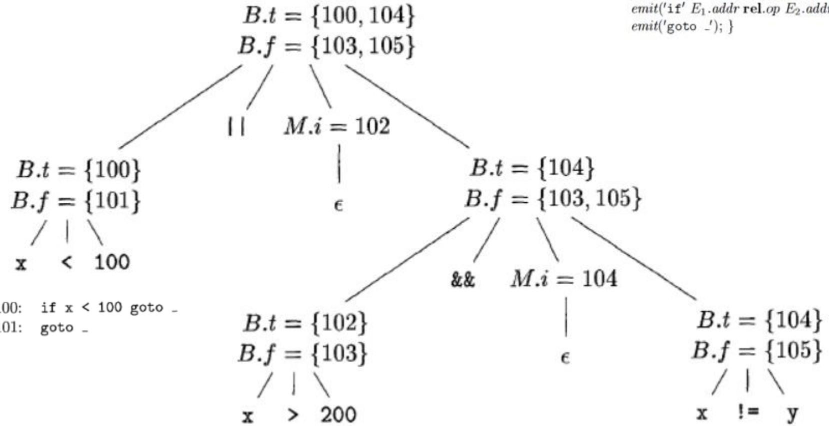
$B \rightarrow B_1 \text{ || } M\ B_2$ { $backpatch(B_1.falselist, M.instr);$
$B.truelist = merge(B_1.truelist, B_2.truelist);$
$B.falselist = B_2.falselist;$ }

$B \rightarrow B_1 \text{ \&\& } M\ B_2$ { $backpatch(B_1.truelist, M.instr);$
$B.truelist = B_2.truelist;$
$B.falselist = merge(B_1.falselist, B_2.falselist);$ }

$B \rightarrow E_1 \textbf{ rel } E_2$ { $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$emit($'if' $E_1.addr$ **rel**.op $E_2.addr$ 'goto _'$);$
$emit($'goto _'$);$ }

Consider again the expression

$$x < 100 \text{ || } x > 200 \text{ \&\& } x \text{ != } y$$




```
100:  if x < 100 goto _
101:  goto _
```

```
102:  if x > 200 goto _
103:  goto _
```

Consider again the expression

$$x < 100 \;||\; x > 200 \;\&\&\; x \;! = \; y$$

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto
103:  goto _
104:  if x != y goto _
105:  goto _
```

Consider again the expression

$$x < 100 \;||\; x > 200 \;\&\&\; x \;!=\; y$$

```
100:   if x < 100 goto _
101:   goto _
102:   if x > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

(a) After backpatching 104 into instruction 102.

```
100:   if x < 100 goto _
101:   goto 102
102:   if y > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

(b) After backpatching 102 into instruction 101.

The entire expression is true if and only if the gotos of instructions 100 or 104 are reached, and is false if and only if the gotos of instructions 103 or 105 are reached. These instructions will have their targets filled in later in the compilation, when it is seen what must be done depending on the truth or falsehood of the expression.

Flow-of-Control Statements

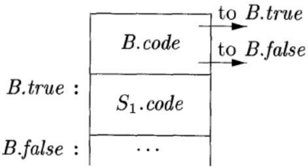
Boolean expressions generated by nonterminal B have two lists of jumps, **B.truelist and B.falselist**, corresponding to the true and false exits from the **code for B**

Statements generated by **nonterminals S and L** have a list of **unfilled jumps**

S.nextlist is a list of **all conditional and unconditional jumps** to the instruction following the code for **statement S** in execution order. **L.nextlist** is defined similarly.

Flow-of-Control Statements

1) $S \rightarrow \textbf{if}\,(\,B\,)\; M\; S_1$	$\{\; backpatch(B.truelist,\; M.instr);$ $S.nextlist \;=\; merge(B.falselist,\; S_1.nextlist);\; \}$
2) $S \rightarrow \textbf{if}\,(\,B\,)\; M_1\; S_1\; N\; \textbf{else}\; M_2\; S_2$	$\{\; backpatch(B.truelist,\; M_1.instr);$ $backpatch(B.falselist,\; M_2.instr);$ $temp \;=\; merge(S_1.nextlist,\; N.nextlist);$ $S.nextlist \;=\; merge(temp,\; S_2.nextlist);\; \}$
3) $S \rightarrow \textbf{while}\; M_1\; (\,B\,)\; M_2\; S_1$	$\{\; backpatch(S_1.nextlist,\; M_1.instr);$ $backpatch(B.truelist,\; M_2.instr);$ $S.nextlist \;=\; B.falselist;$ $emit(\text{'goto'}\; M_1.instr);\; \}$
4) $S \rightarrow \{\; L\; \}$	$\{\; S.nextlist \;=\; L.nextlist;\; \}$
5) $S \rightarrow A\; ;$	$\{\; S.nextlist \;=\; \textbf{null};\; \}$
6) $M \rightarrow \epsilon$	$\{\; M.instr \;=\; nextinstr;\; \}$
7) $N \rightarrow \epsilon$	$\{\; N.nextlist \;=\; makelist(nextinstr);$ $emit(\text{'goto \_'});\; \}$
8) $L \rightarrow L_1\; M\; S$	$\{\; backpatch(L_1.nextlist,\; M.instr);$ $L.nextlist \;=\; S.nextlist;\; \}$
9) $L \rightarrow S$	$\{\; L.nextlist \;=\; S.nextlist;\; \}$



(a) if

Flow-of-Control Statements

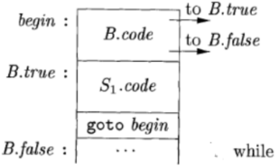
Homework

```
if(x<100||x>200 && x!=y)
    a=0;
b=0;
```

Flow-of-Control Statements (while)

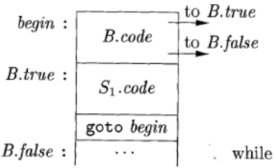
$$S \rightarrow \textbf{while} \ (B) \ S_1$$

$$S \rightarrow \textbf{while} \ M_1 \ (B) \ M_2 \ S_1$$
$$\{ \ backpatch(S_1.nextlist, \ M_1.instr);$$
$$backpatch(B.truelist, \ M_2.instr);$$
$$S.nextlist \ = \ B.falselist;$$
$$emit('\texttt{goto}' \ M_1.instr); \ \}$$



- The two occurrences of the **marker nonterminal M** record the instruction numbers of the
- M1--beginning of the code for B (**begin**) and the
- M2--beginning of the code for S1 (**B.true**).

Flow-of-Control Statements (while)



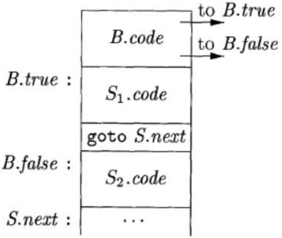
$$
\begin{aligned}
S \rightarrow\ & \textbf{while}\ M_1\ (\ B\ )\ M_2\ S_1 \\
& \{\ backpatch(S_1.nextlist,\ M_1.instr); \\
& \ \ backpatch(B.truelist,\ M_2.instr); \\
& \ \ S.nextlist\ =\ B.falselist; \\
& \ \ emit('\texttt{goto}'\ M_1.instr);\ \}
\end{aligned}
$$

- Attribute **M.instr** points to the number of the **next instruction**.
- After the **body S1 of the while-statement is executed**,
    - Control flows to the beginning.
    - We backpatch **S1.nextlist** to make all targets on that list be **M1.instr**.
- An **explicit jump** to the beginning of the code for B is appended after the code for S1

- **B. truelist** is backpatched to go to the **beginning of S1** ‘
- by making jumps on **B.truelist** go to **M2.instr**

Flow-of-Control Statements (if-else)

if (B) S_1 **else** S_2

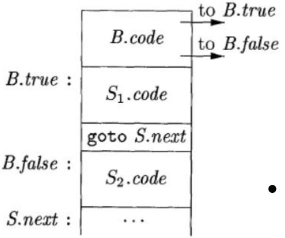


$$
\begin{array}{lll}
S \rightarrow & \textbf{if}\,(\,B\,)\; M_1\; S_1\; N\; \textbf{else}\; M_2\; S_2 & \\
& & \{\; backpatch(B.truelist,\; M_1.instr); \\
& & \;\; backpatch(B.falselist,\; M_2.instr); \\
& & \;\; temp \;=\; merge(S_1.nextlist,\; N.nextlist); \\
& & \;\; S.nextlist \;=\; merge(temp,\; S_2.nextlist);\; \} \\
N \rightarrow \epsilon & & \{\; N.nextlist \;=\; makelist(nextinstr); \\
& & \;\; emit('\texttt{goto}\;\_');\; \}
\end{array}
$$

We backpatch the jumps when B is true to the instruction $M_1.instr$; the latter is the beginning of the code for S_1. Similarly, we backpatch jumps when B is false to go to the beginning of the code for S_2. The list $S.nextlist$ includes all jumps out of S_1 and S_2, as well as the jump generated by N. (Variable *temp* is a temporary that is used only for merging lists.)

Flow-of-Control Statements (if-else)

$\textbf{if} (B) S_1 \textbf{ else } S_2$



$$S \rightarrow \textbf{if} (B) M_1 S_1 N \textbf{ else } M_2 S_2$$
$$\{ backpatch(B.truelist, M_1.instr);$$
$$backpatch(B.falselist, M_2.instr);$$
$$temp = merge(S_1.nextlist, N.nextlist);$$
$$S.nextlist = merge(temp, S_2.nextlist); \}$$

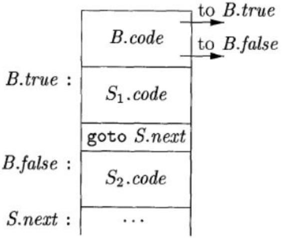
$$N \rightarrow \epsilon \qquad \{ N.nextlist = makelist(nextinstr);$$
$$emit(\text{'goto \_'}); \}$$

- **S1 is an assignment statement**, we must include at the end of the code for S1 a **jump over the code for S2**

- N has attribute **N.nextlist**, which will be a list consisting of the instruction number of the **jump goto _**

Flow-of-Control Statements (if-else)

if (B) S_1 **else** S_2



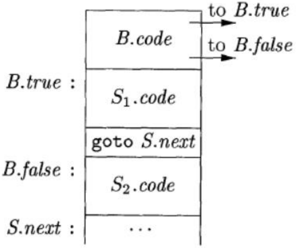
$$S \rightarrow \textbf{if}\,(\,B\,)\; M_1\; S_1\; N\; \textbf{else}\; M_2\; S_2$$
$$\{\; backpatch(B.truelist,\; M_1.instr);$$
$$backpatch(B.falselist,\; M_2.instr);$$
$$temp \;=\; merge(S_1.nextlist,\; N.nextlist);$$
$$S.nextlist \;=\; merge(temp,\; S_2.nextlist);\; \}$$

$$N \rightarrow \epsilon \qquad \{\; N.nextlist \;=\; makelist(nextinstr);$$
$$emit('\texttt{goto \_}');\; \}$$

We backpatch the jumps when B is true to the instruction $M_1.instr$; the latter is the beginning of the code for S_1. Similarly, we backpatch jumps when B is false to go to the beginning of the code for S_2. The list $S.nextlist$ includes all jumps out of S_1 and S_2, as well as the jump generated by N. (Variable $temp$ is a temporary that is used only for merging lists.)

Translation of if-else statement

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
$S \rightarrow \textbf{if} \; (\; B \;) \; S_1 \; \textbf{else} \; S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\|\| \; label(B.true) \;\|\|\; S_1.code$ $\|\| \; gen('\texttt{goto}' \; S.next)$ $\|\| \; label(B.false) \;\|\|\; S_2.code$



(b) if-else

Intermediate Code for Procedures

$$
\begin{array}{lcl}
D & \rightarrow & \textbf{define}\ T\ \textbf{id}\ (\ F\ )\ \{\ S\ \} \\
F & \rightarrow & \epsilon\ \mid\ T\ \textbf{id}\ ,\ F \\
S & \rightarrow & \textbf{return}\ E\ ; \\
E & \rightarrow & \textbf{id}\ (\ A\ ) \\
A & \rightarrow & \epsilon\ \mid\ E\ ,\ A
\end{array}
$$

- Nonterminals **D and T** generate **function definition and types**, respectively
- A **function definition generated by D** consists of **keyword define**, a return type, the function name, formal parameters in parentheses and a function body consisting of a statement.
- **Nonterminal F generates** zero or more **formal parameters**, where a formal parameter consists of a type followed by an identifier.
- **Nonterminals S and E generate statements and expressions**, respectively. The production for **S adds a statement** that **returns** the value of an expression.
- The **production for E adds function calls**, with actual parameters generated by A. An actual parameter is an expression.

Intermediate Code for Procedures

```
n = f(a[i]);
```

following three-address code:

1) t_1 = i * 4
2) t_2 = a [t_1]
3) param t_2
4) t_3 = call f, 1
5) n = t_3

Function calls.

- When generating three-address instructions for a function call id(E, E,... , E),
- It is sufficient to generate the **three-address instructions for evaluating or reducing the parameters E** to addresses (E.addr),
- Followed by a **param instruction** for each parameter.

Intermediate Code for Procedures

```
n = f(a[i]);
```

following three-address code:

1) t_1 = i * 4
2) t_2 = a [t_1]
3) param t_2
4) t_3 = call f, 1
5) n = t_3

Symbol tables.

- Let **s be the top symbol table** when the function definition is reached.
- The **function name is entered into symbol table s** for use in the rest of the program.
- Data type ***function (return type, parameter type)*** inserted in symbol table

----

- The **formal parameters** of a function can be handled in analogy with field names in a record
- In the production for D, after seeing define and the function name, we push s and set up a new symbol table
- Env.push(top)]
- t = new Env();
- The new symbol table, t.
- The new table t is used to translate the function body.
- We revert to the previous symbol table s after the function body is translated.

Record or Structure data type

$$T \rightarrow \textbf{record} \; '\{' \; D \; '\}'$$

$$\begin{array}{lll} T \rightarrow \textbf{record} \; '\{' & \{ \; Env.push(top); \; top = \textbf{new} \; Env(); \\ & \quad Stack.push(offset); \; offset = 0; \; \} \\ \quad D \; '\}' & \{ \; T.type = record(top); \; T.width = offset; \\ & \quad top = Env.pop(); \; offset = Stack.pop(); \; \} \end{array}$$

For convenience, record types will encode both the types and relative addresses of their fields, using a symbol table for the record type. A record type has the form *record*(t), where *record* is the type constructor, and t is a symbol-table object that holds information about the fields of this record type.

The embedded action before D saves the existing symbol table, denoted by *top* and sets *top* to a fresh symbol table. It also saves the current *offset*, and sets *offset* to 0. The declarations generated by D will result in types and relative addresses being put in the fresh symbol table. The action after D creates a record type using *top*, before restoring the saved symbol table and offset.